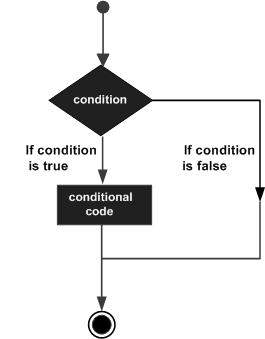
**Decision Control Statements**

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements.

|  |  |
| --- | --- |
| S.No. | Statement & Description |
| 1 | if statements  An if statement consists of a Boolean expression followed by one or more statements. |
| 2 | if...else statements  An if statement can be followed by an optional else statement, which executes when the Boolean expression is FALSE. |
| 3 | nested if statements  You can use one if or else if statement inside another if or else ifstatement(s). |

**Simple if:**

If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a one-line if clause:

#!/usr/bin/python

var = 100

if ( var == 100 ) :

print "Value of expression is 100"

print "Good bye!"

When the above code is executed, it produces the following result:

Value of expression is 100

Good bye!

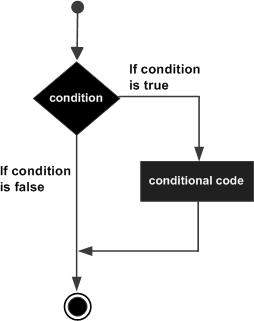
It is similar to that of other languages. The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

**Syntax:**

if expression:

statement(s)

If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If Boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.



**Example:**

#!/usr/bin/python

var1 = 100

if var1:

print "1 - Got a true expression value"

print var1

var2 = 0

if var2:

print "2 - Got a true expression value"

print var2

print "Good bye!"

When the above code is executed, it produces the following result:

1 - Got a true expression value

100

Good bye!

**If….Else:**

An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else statement following if.

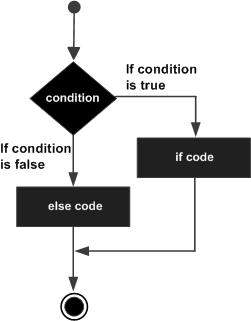
**Syntax**

if expression:

statement(s)

else:

statement(s)



**Example:**

#!/usr/bin/python

var1 = 100

if var1:

print "1 - Got a true expression value"

print var1

else:

print "1 - Got a false expression value"

print var1

var2 = 0

if var2:

print "2 - Got a true expression value"

print var2

else:

print "2 - Got a false expression value"

print var2

print "Good bye!"

When the above code is executed, it produces the following result:

1 - Got a true expression value

100

2 - Got a false expression value

0

Good bye!

**The elif Statement**

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

**Syntax:**

if expression1:

statement(s)

elif expression2:

statement(s)

elif expression3:

statement(s)

else:

statement(s)

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows:

**Example:**

#!/usr/bin/python

var = 100

if var == 200:

print "1 - Got a true expression value"

print var

elif var == 150:

print "2 - Got a true expression value"

print var

elif var == 100:

print "3 - Got a true expression value"

print var

else:

print "4 - Got a false expression value"

print var

print "Good bye!"

When the above code is executed, it produces the following result:

3 - Got a true expression value

100

Good bye!

**Nested If:**

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested ifconstruct.

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

**Syntax**

if expression1:

statement(s)

if expression2:

statement(s)

elif expression3:

statement(s)

else:

statement(s)

elif expression4:

statement(s)

else:

statement(s)

**Example**

#!/usr/bin/python

var = 100

if var < 200:

print "Expression value is less than 200"

if var == 150:

print "Which is 150"

elif var == 100:

print "Which is 100"

elif var == 50:

print "Which is 50"

elif var < 50:

print "Expression value is less than 50"

else:

print "Could not find true expression"

print "Good bye!"

When the above code is executed, it produces following result:

Expression value is less than 200

Which is 100

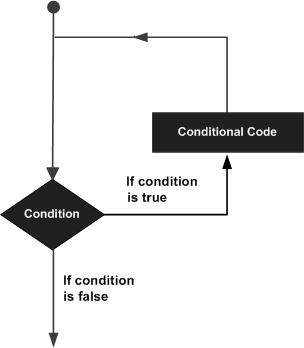
Good bye!

**Loops:**

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **S.No.** | **Loop Type & Description** |
| 1 | while loop  Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | for loop  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | nested loops  You can use one or more loop inside any another while, for or do..while loop. |

**Loop Control Statements**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

|  |  |
| --- | --- |
| **S.No.** | **Control Statement & Description** |
| 1 | break statement  Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | continue statement  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | pass statement  The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

**While loop**

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

**Syntax:**

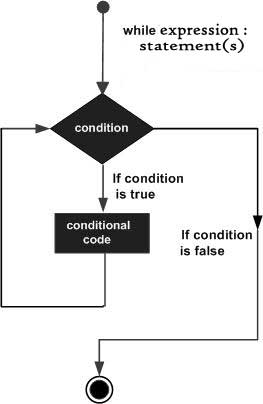
while expression:

statement(s)

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example:**

#!/usr/bin/python

count = 0

while (count < 9):

print 'The count is:', count

count = count + 1

print "Good bye!"

When the above code is executed, it produces the following result:

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

**The Infinite Loop**

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

#!/usr/bin/python

var = 1

while var == 1 : # This constructs an infinite loop

num = raw\_input("Enter a number :")

print "You entered: ", num

print "Good bye!"

When the above code is executed, it produces the following result:

Enter a number :20

You entered: 20

Enter a number :29

You entered: 29

Enter a number :3

You entered: 3

Enter a number between :Traceback (most recent call last):

File "test.py", line 5, in <module>

num = raw\_input("Enter a number :")

**Keyboard Interrupt**

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

**Using else Statement with Loops**

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

#!/usr/bin/python

count = 0

while count < 5:

print count, " is less than 5"

count = count + 1

else:

print count, " is not less than 5"

When the above code is executed, it produces the following result:

0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

5 is not less than 5

**Single Statement Suites (for loop)**

Similar to the if statement syntax, if your while clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a one-line while clause −

#!/usr/bin/python

flag = 1

while (flag):

print 'Given flag is really true!'

print "Good bye!"

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

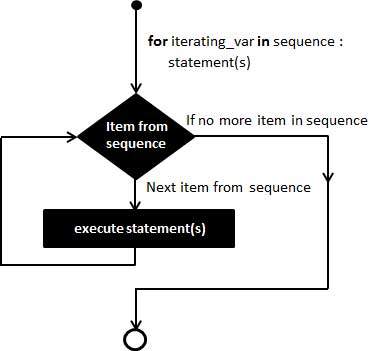
It has the ability to iterate over the items of any sequence, such as a list or a string.

**Syntax**

for iterating\_var in sequence:

statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating\_var. Next, the statements block is executed. Each item in the list is assigned to iterating\_var, and the statement(s) block is executed until the entire sequence is exhausted.



**Example**

#!/usr/bin/python

for letter in 'Python': # First Example

print 'Current Letter :', letter

fruits = ['banana', 'apple', 'mango']

for fruit in fruits: # Second Example

print 'Current fruit :', fruit

print "Good bye!"

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

**# Program to find the sum of all numbers stored in a list**

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

sum = sum+val

# Output: The sum is 48

print("The sum is", sum)

**Iterating by Sequence Index**

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example:

#!/usr/bin/python

fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

print 'Current fruit :', fruits[index]

print "Good bye!"

When the above code is executed, it produces the following result −

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

**Using else Statement with Loops**

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

#!/usr/bin/python

for num in range(10,20): #to iterate between 10 to 20

for i in range(2,num): #to iterate on the factors of the number

if num%i == 0: #to determine the first factor

j=num/i #to calculate the second factor

print '%d equals %d \* %d' % (num,i,j)

break #to move to the next number, the #first FOR

else: # else part of the loop

print num, 'is a prime number'

When the above code is executed, it produces the following result −

10 equals 2 \* 5

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

11 is a prime number

12 equals 2 \* 6

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

13 is a prime number

14 equals 2 \* 7

15 is a prime number

15 equals 3 \* 5

16 equals 2 \* 8

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

17 is a prime number

18 equals 2 \* 9

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

19 is a prime number

**Nested Loops**

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

**Syntax**

for iterating\_var in sequence:

for iterating\_var in sequence:

statements(s)

statements(s)

The syntax for a nested while loop statement in Python programming language is as follows −

while expression:

while expression:

statement(s)

statement(s)

A final note on loop nesting is that you can put any type of loop inside of any other type of loop.

For example a for loop can be inside a while loop or vice versa.

**Example**

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

#!/usr/bin/python

i = 2

while(i < 100):

j = 2

while(j <= (i/j)):

if not(i%j): break

j = j + 1

if (j > i/j) : print i, " is prime"

i = i + 1

print "Good bye!"

When the above code is executed, it produces following result −

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

Good bye!

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

**Break Statement:**

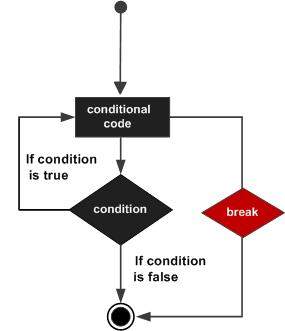
The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

**Syntax**

The syntax for a break statement in Python is as follows −

break



**Example**

#!/usr/bin/python

for letter in 'Python': # First Example

if letter == 'h':

break

print 'Current Letter :', letter

var = 10 # Second Example

while var > 0:

print 'Current variable value :', var

var = var -1

if var == 5:

break

print "Good bye!"

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current variable value : 10

Current variable value : 9

Current variable value : 8

Current variable value : 7

Current variable value : 6

Good bye!

It returns the control to the beginning of the while loop..

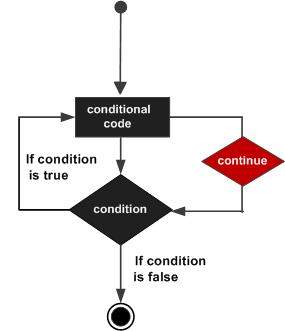
**Continue Statement:**

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.

**Syntax**

continue



**Example**

#!/usr/bin/python

for letter in 'Python': # First Example

if letter == 'h':

continue

print 'Current Letter :', letter

var = 10 # Second Example

while var > 0:

var = var -1

if var == 5:

continue

print 'Current variable value :', var

print "Good bye!"

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

Current variable value : 9

Current variable value : 8

Current variable value : 7

Current variable value : 6

Current variable value : 4

Current variable value : 3

Current variable value : 2

Current variable value : 1

Current variable value : 0

Good bye!

It is used when a statement is required syntactically but you do not want any command or code to execute.

**Pass statement:**

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet.

**Syntax**

pass

**Example**

#!/usr/bin/python

for letter in 'Python':

if letter == 'h':

pass

print 'This is pass block'

print 'Current Letter :', letter

print "Good bye!"

When the above code is executed, it produces following result −

Current Letter : P

Current Letter : y

Current Letter : t

This is pass block

Current Letter : h

Current Letter : o

Current Letter : n

Good bye!

**The range() function:**

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

# Output: range(0, 10)

print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]

print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]

print(list(range(2, 20, 3)))

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing.

**# Program to iterate through a list using indexing**

genre = ['pop', 'rock', 'jazz']

**# iterate over the list using index**

for i in range(len(genre)):

print("I like", genre[i])

**# Python program to illustrate**

**# Iterating over a list**

print("List Iteration")

l = ["geeks", "for", "geeks"]

for i in l:

     print(i)

**# Iterating over a tuple (immutable)**

print("\nTuple Iteration")

t = ("geeks", "for", "geeks")

for i in t:

     print(i)

**# Iterating over a String**

print("\nString Iteration")

s = "Geeks"

for i in s :

     print(i)

**# Iterating over dictionary**

print("\nDictionary Iteration")

d = dict()

d['xyz'] = 123

d['abc'] = 345

for i in d :

     print("%s  %d" %(i, d[i]))

**Iterating by index of sequences:**

We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and in iterate over the sequence within the range of this length.  
See the below example:

|  |
| --- |
| **# Python program to illustrate**  **# Iterating by index**    list = ["geeks", "for", "geeks"]  for index in range(len(list)):      print list[index] |

**Using else statement with for loops:**

We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

|  |
| --- |
| # Python program to illustrate  # combining else with for    list = ["geeks", "for", "geeks"]  for index in range(len(list)):      print list[index]  else:      print "Inside Else Block" |

**Nested Loops:**

Python programming language allows to use one loop inside another loop.

**Syntax:**

|  |
| --- |
| for iterator\_var in sequence:      for iterator\_var in sequence:          statements(s)          statements(s) |

**Syntax:**

|  |
| --- |
| while expression:      while expression:          statement(s)          statement(s) |

A final note on loop nesting is that we can put any type of loop inside of any other type of loop.

For example a for loop can be inside a while loop or vice versa.

|  |
| --- |
| # Python program to illustrate  # nested for loops in Python  from \_\_future\_\_ import print\_function  for i in range(1, 5):      for j in range(i):           print(i, end=' ')      print() |
|  |

Output:

1

2 2

3 3 3

4 4 4 4